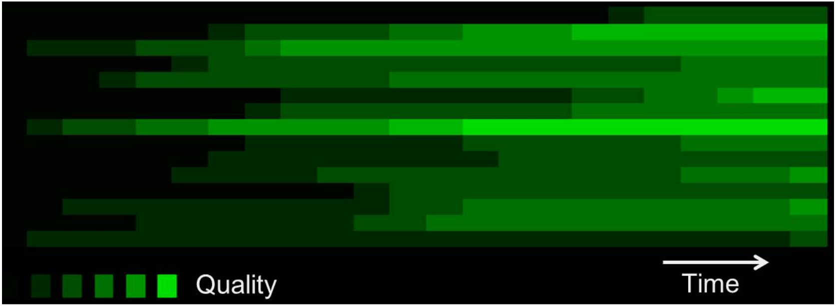
Flexible Requirements Modeling with reqT – a hands-on tutorial

http://reqT.org

bjornregnell.se, Lund University, Sweden

12th Swedish Requirements Engineering Research Network Signal,
SiREN2013 sirensweden.org

Good enough Requirements Engineering?



3 software engineering trends: Decentralize, Distribute, Document less

- Agile teams + SW ecosystems -> No centrally controlled, detailed "master plan"
- Continuous integration & deployment
- Increased parallelization
- Distributed Version Control, e.g. Git
- "Code is king"

A challenge for the RE communitty

How to best help code-focused, agile software engineers to do good enough requirements engineering in a decentralized, distributed, documentation-hostile setting?

Goals of reqT – a Requirements Engineering Tool

Design an interesting tool based on these goals:

Goal	Design choices	Rationale
Semi-formal	• Use graph structures • Mix human Natural Language (NL) with essential RE semantics	• Graphs are well-known by software engineers and powerful for expressing structure and flexible for search. • NL is well-known and powerful...
Open	• Free, permissive OSS license • Cross-platform: JVM	• Allow integration of existing code bases in JVM-based languages • Enable academic usage and contribution in teaching and research
Scalable	• Internal DSL in Scala `www.scala-lang.org`	• Open-ended language • Scala is scalable, powerful, concise, typesafe, scriptable, ...

Short paper published at REFSQ2013: `http://www.reqt.org/reqT-REFSQ2013-paper.pdf`

reqT: Requirements => ComputationalEntities

In reqT ...

- requirements are computational entities
- requirements are serialized as self-generating code
- the meta-model and its semantics are flexible:

```
> allow me to mix text with structure
> warn me, don't force me
```

What can you do with reqT?

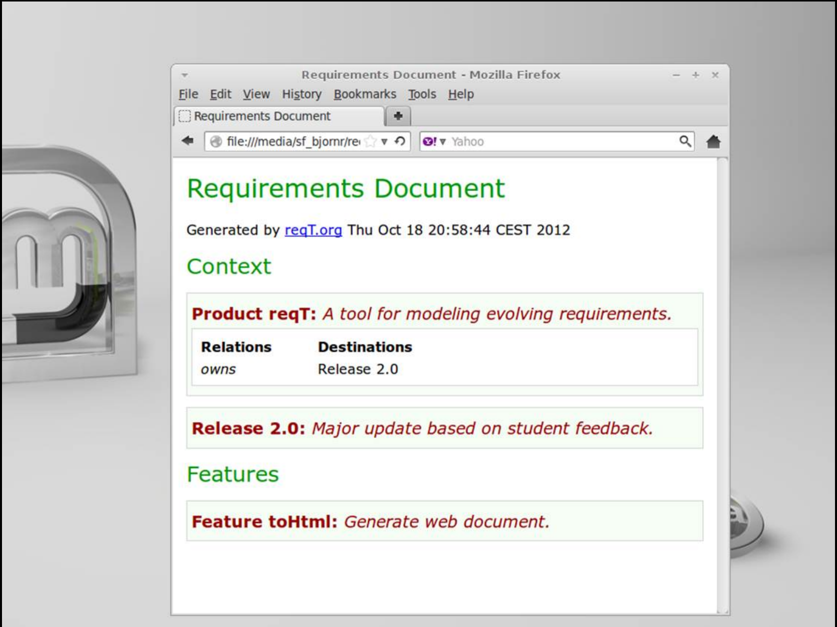
- Create and manage requirements models using versatile collections
- Combine natural language expressiveness with type-safe modeling
- Interoperate with spread sheet applications
- Auto-generate requirements documents for web publishing
- Do powerful scripting of requirements models with the Scala-embedded DSL
- Model and solve combinatorial decision problems in RE such as Prioritisation and Release Planning using a sub-DSL for Constraint Satisfaction Programming wrapping the JaCoP solver

`http://reqT.org`

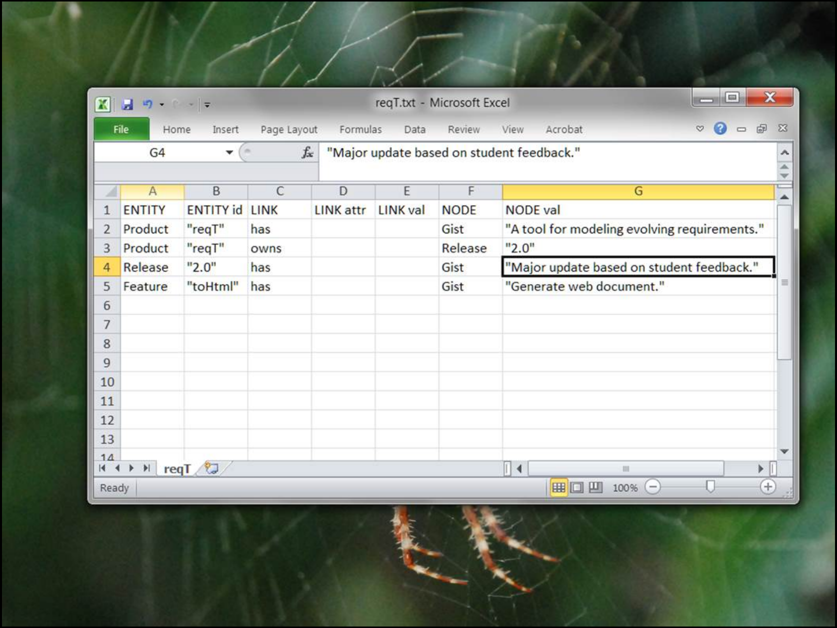
`http://www.jacop.eu`

reqT Gist

```
var m = Model(
  Product("reqT") has Gist("A tool for modeling evolving requirements."),
  Release("2.0") has Gist("Major update based on student feedback."),
  Release("2.3") has Gist("Constraint solving for decision problems."),
  Product("reqT") owns (Release("2.0"), Release("2.3"))
)

m += Feature("toHtml") has Gist("Generate web document.")

println(m)
m.toHtml.save("reqT.html")
m.toTable.save("reqT.txt")
```



Requirements Document - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Requirements Document
file:///media/sf_bjornr/re
Yahoo
Requirements Document
Generated by reqT.org Thu Oct 18 20:58:44 CEST 2012
Context
Product reqT: A tool for modeling evolving requirements.
Relations Destinations
owns Release 2.0
Release 2.0: Major update based on student feedback.
Features
Feature toHtml: Generate web document.



	A	B	C	D	E	F	G
1	ENTITY	ENTITY id	LINK	LINK attr	LINK val	NODE	NODE val
2	Product	"reqT"	has			Gist	"A tool for modeling evolving requirements."
3	Product	"reqT"	owns			Release	"2.0"
4	Release	"2.0"	has			Gist	"Major update based on student feedback."
5	Feature	"toHtml"	has			Gist	"Generate web document."

reqT provides a DSL for RE embedded in Scala

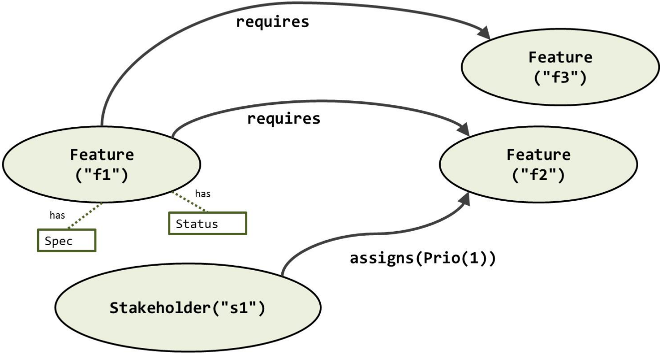
A reqT model includes a sequence of graph parts
<Entity> <Edge> <NodeSet>
separated by comma and wrapped inside a Model( )

```
var myRequirements =  Model(
  Feature("f1") has (Spec("A good spec."), Status(SPECIFIED)),
  Feature("f1") requires (Feature("f2"), Feature("f3")),
  Stakeholder("s1") assigns(Prio(1)) to Feature("f2")
)
```

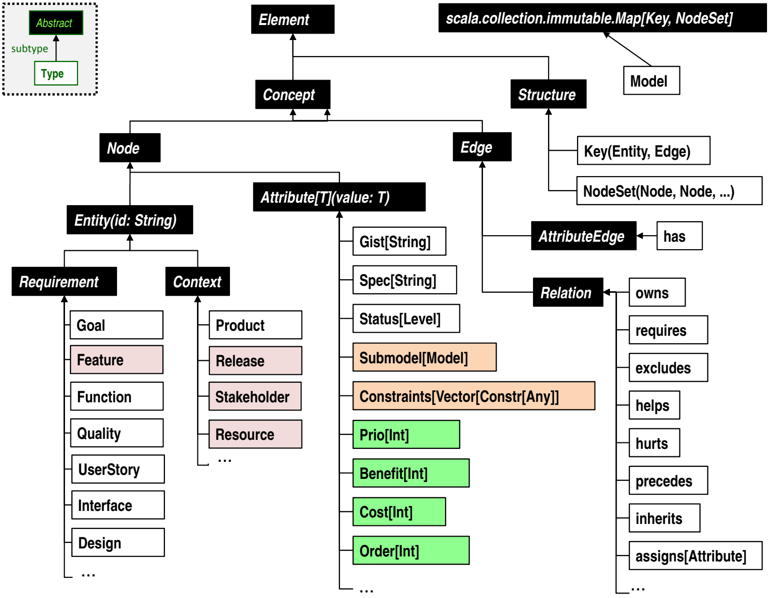
Download: `http://reqT.org`
Source code: `https://github.com/reqT/reqT`

reqT models are graph structures with Entities & Attributes (nodes) and Relations (edges)

```
var myRequirements =  Model(
  Feature("f1") has (Spec("A good spec."), Status(SPECIFIED)),
  Feature("f1") requires (Feature("f2"), Feature("f3")),
  Stakeholder("s1") assigns(Prio(1)) to Feature("f2")
)
```



Overview of the reqT metamodel



reqT's all entities, attributes and relations (v2.3RC1)

Entities:

Context
Product, Release,
Stakeholder, Actor,
Resource, Subdomain

Requirement
Req, Idea, Goal,
Feature, Function,
Quality, Interface,
Design,
Issue, Ticket

Scenario
UserStory, UseCase,
TestCase, Task,
VividScenario

Data
Class, Member

Relations:

owns,
requires,
excludes,
releases,
helps, hurts,
precedes,
inherits,
implements,
verifies,
deprecates,
assigns

Attributes:

String values:
Gist, Spec, Why, Example,
Input, Output, Expectation,
Trigger, Precond, Frequency,
Critical, Problem, Label,
Comment, Image,
Deprecated, Code,

Level values:
Status,

Integer values:
Prio, Order, Cost, Benefit,
Capacity, Urgency,

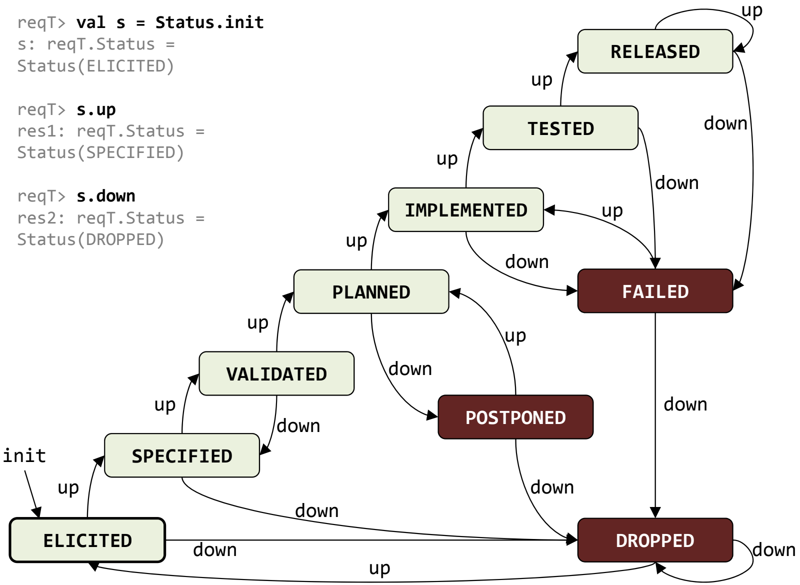
collection values...
Submodel, Constraints

reqT models can contain code and execute test cases

```
val m1 = Model(
  TestCase("add1") has (Code("1 + 42"), Expectation("42")),
  TestCase("add2") has (Code("{1 to 42}.sum"), Expectation("903")),
  TestCase("mul1") has (External[Code]("filename.scala"), Expectation("true"))
)

reqT> m1.loadExternals.run
res1: scala.collection.immutable.Map[reqt.Entity,String] =
Map(TestCase(add1) -> 43, TestCase(add2) -> 903, TestCase(mul1) -> true)

reqT> (m1 / "add2").tested
res2: reqt.Model =
Model(
  TestCase("add2") has (Expectation("903"), Output("903"), Code("{1 to 42}.sum"))
)

reqT> m1.loadExternals.isTestOk
*** FAILED: TestCase(add1)
  Output:    43
  Expectation: 42
res3: Boolean = false
```

reqT Level values of the Status attribute

```
reqT> val s = Status.init
s: reqT.Status =
Status(ELICITED)

reqT> s.up
res1: reqT.Status =
Status(SPECIFIED)

reqT> s.down
res2: reqT.Status =
Status(DROPPED)
```



reqT update of Status attribute

```
reqT> var m = Model(Feature("x") has Status.init, Feature("y") has Status.init)
m: reqT.Model =
Model(
  Feature("x") has Status(ELICITED),
  Feature("y") has Status(ELICITED)
)

reqT> m.up
res8: reqT.Model =
Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(SPECIFIED)
)

reqT> m = (m / Feature("x")).up ++ (m \ Feature("x"))
m: reqT.Model = Model(
  Feature("x") has Status(SPECIFIED),
  Feature("y") has Status(ELICITED)
)

reqT> m = m up Feature("x")          //equivalent shorter way to do previous
```

reqT models can be hierarchical with recursive submodels in a tree structure

```
var myReqs = Model(
  Feature("nice") has Spec("this is a nice feature"),
  Feature("cool") has Spec("this is a cool feature"),
  Stakeholder("Tony") has Submodel(
    Feature("nice") has Prio(1),
    Feature("cool") has Prio(2)
  ),
  Stakeholder("Annabella") has Submodel(
    Feature("nice") has Prio(2),
    Feature("cool") has Prio(1)
  )
)
```

reqT can reference values of attributes in deeply nested submodel structures using the ! operator

```
(Feature("f")!Prio)
(Stakeholder("a")!Feature("g")!Benefit)
```

```
val m = Model(
  Feature("f") has Prio(1),
  Stakeholder("a") has Submodel(
    Feature("g") has Benefit(2),
    Resource("x") has Submodel(
      Feature("h") has Cost(3)
    )
  )
)

m(Feature("f")!Prio) == 1
m(Stakeholder("a")!Feature("g")!Benefit) == 2
m(Stakeholder("a")!Resource("x")!Feature("h")!Cost) == 3
```

reqT context description example

You can use reqT with your favourite RE text book.

```
Model(
  Product("Hotel system") owns (
    Interface("ReceptionUI"), Interface("GuestUI"),
    Interface("TelephonyAPI"), Interface("AccountingAPI")
  ),
  Product("Hotel system") has Image("context-diagram.png"),
  Interface("ReceptionUI") has (
    Input("booking, check-out"), Output("service note"),
    Image("receptionUI-screen.png")
  ),
  Interface("GuestUI") has (
    Output("confirmation, invoice"),
    Image("guestUI-screen.png")),
  Actor("Receptionist") requires Interface("ReceptionUI"),
  Actor("Guest") requires Interface("GuestUI"),
  Actor("Receptionist") requires Interface("ReceptionUI"),
  Actor("Telephony System") requires Interface("TelephonyAPI"),
  Actor("Accounting System") requires Interface("AccountingAPI")
)
```

[Example modified from Lauesen: Software Requirements – Styles and Techniques]

reqT task description example

```
Model(
  Task("reception work") owns (Task("check in"), Task("booking")),
  Task("check in") has (
    Why("Give guest a room. Mark it as occupied. Start account."),
    Trigger("A guest arrives"),
    Frequency("Average 0.5 checkins/room/day"),
    Critical("Group tour with 50 guests.")
  ),
  Task("check in") owns (
    Task("find room"), Task("record guest"), Task("deliver key")),
  Task("record guest") has Spec(
    "variants: a) Guest has booked in advance, b) No suitable room"
  )
)
```

[Example modified from Lauesen: Software Requirements – Styles and Techniques]

reqT quality requirements example

```
Model(
  Quality("capacity database") has
    Spec("#guests < 10,000 growing 20% per year, #rooms < 1,000"),
  Quality("accuracy calendar") has
    Spec("Bookings shall be possible at least two years ahead."),
  Quality("performance forecast") has
    Spec("Product shall compute a room occupation forecast
          within ___ minutes. (Customer expects one minute.)"),
  Quality("usability task time") has
    Spec("Novice users shall perform tasks Q and R in 15 minutes.
          Experienced users tasks Q, R, S in 2 minutes."),
  Quality("usability task time") requires (Task("Q"), Task("R"), Task("S"))
  Quality("performance peak load") has
    Spec("Product shall be able to process 100 payment transactions
          per second in peak load.")
)
```

[Example modified from Lauesen: Software Requirements – Styles and Techniques]

reqT QUPER example

```
Model(
  UserStory("playMusic") has Spec("As a user I want to play music."),
  UserStory("playMusic") requires Quality("timeToMusic"),
  Quality("performance") owns (Quality("timeToMusic"), Quality("timeToVideo")),
  Quality("timeToMusic.metric") has Spec("Measured in seconds using tests XYZ."),
  Quality("timeToMusic.ref.X") has (Spec("4.0 s"), Comment("Competitor product X.")),
  Quality("timeToMusic.ref.Y") has (Spec("2.0 s"), Comment("Competitor product Y.")),
  Quality("timeToMusic.ref.Z") has (Spec("3.0 s"), Comment("Our own released product Z.")),
  Quality("timeToMusic.utility") has Spec("5.0 s"),
  Quality("timeToMusic.differentiation") has Spec("1.5 s"),
  Quality("timeToMusic.saturation") has Spec("0.2 s"),
  Quality("timeToMusic.barrier.1") has Spec("2.0 s requires Effort(Range(4,5),Weeks)"),
  Quality("timeToMusic.barrier.2") has Spec("1.0 s requires Effort(Range(24,48),Weeks)"),
  Quality("timeToMusic.target.min") has (Spec("2.0 s"), Comment("Probably possible with existing architecture.")),
  Quality("timeToMusic.target.max") has (Spec("1.0 s"), Comment("Probably needs new architecture.")),
  Quality("timeToMusic") has Image("QUPER-timeToMusic.jpg")
)
```

Example modified from "Setting quality targets for coming releases with QUPER: an industrial case study", R. Berntsson Svensson, Y. Sprockel, B. Regnell, S. Brinkkemper, Requirements Engineering, November 2012, Volume 17, Issue 4, pp 283-298

reqT some example operations on Models

To do this...	...code this...
Create empty model	`var m = Model()`
Add entity with one attribute [1]	`m += Feature("hello") has Spec("print da stuff")`
Add entity with two attributes	`m += Feature("f1") has (Gist("g1"), Spec("s1"))`
Overwriting existing attribute	`m += Feature("f1") has Gist("g2")`
Add an owns-relation [2]	`m += Product("p1") owns (Feature("f1"), Feature("hello"))`
Remove an entity [3]	`var m2 = m - Feature("f1")`

Restrict operator	`m / Feature("f1")`
	`m / Feature`
	`m / Spec`
	`m / Context`
	`m / Feature / Gist`

Restrict to destinations	`m /-> Feature("f1")`
Extended restrict adds destinations	`m /+ Feature("f1")`
Depth first search	`m /--> Product("p1")`

[1] m += x is the same as m = m + x

[2] owns is a special relation: an entity can only have one direct owner

[3] all operations are immutable; new model is created

reqT set operations, complement to restrictions, etc.

To do this...	...code this...

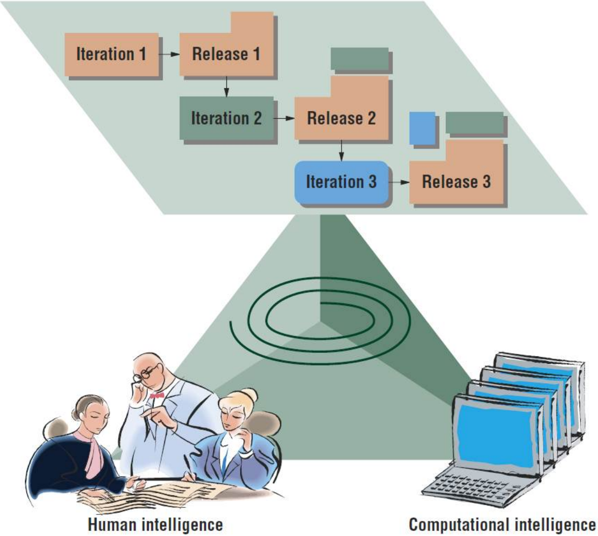
Partition	`var (mx, my) = m | Feature`
	`var (mx, my) = m |+ Feature`
	`var (mx, my) = m |-> Feature`
	`var (mx, my) = m |--> Feature`

Aggregate	`mx ++ my`
Difference	`mx -- my`
Intersect	`mx & my`
Exclude	`m \ Feature("f1")`

Other Complement operators	`m \-> Feature("f1")`
	`m \+ Feature("f1")`
	`m \--> Feature("f1")`

Add same attribute to all entities	`m + Gist("same same")`
Remove all Gist attributes	`m - Gist`

Release Planning in Software Develepment



[Ruhe et al.]

Why Constraint Solving in Requirements Engineering?

Some potential benefits of CSP in RE:

- Flexible specification of decision problems
  - Prioritization
  - Release Planning
- Interactive exploration of the solution space
- Out-of-the-box optimization support

Some challenges:

- How to integrate CSP with RE technology and make it user friendly in the domain?
- How to model CSP problems at the right abstraction level given great uncertainties?

Constraint-based **Priority Ranking** example: 5 features ranked from 1 to 5

reqT:

```
val n = 5
var f = vars(n, "f")
val Result(conclusion, nSol, sol, _ , _) =
  Constraints(
    f::{0 until n},
    AllDifferent(f),
    f(0) #> f(1),
    f(1) #> f(2),
    f(2) #< f(3),
    forAll(0 until n) { f(4) #>= f(_) }
  ).solve(Satisfy)
```

MiniZinc:

```
int: n = 5;
array[1..n] of var 1..n: f;
constraint
  alldifferent(f);
constraint f[1] > f[2];
constraint f[2] > f[3];
constraint f[3] < f[4];
constraint
  forall ( i in 1..n)
    ( f[5] >= f[i] );
solve satisfy;
```

reqT CSP: parameters to solve

Objective parameter

`Satisfy`	find one solution (if any)
`CountAll`	count the number of solutions
`FindAll`	record all solutions
`Maximize(Var("x"))` `Manimize(Var("x"))`	find the solution (if any) that is optimal

Other optional parameters

`timeOutOption`	limits the time of the search (seconds)
`solutionLimitOption`	limits the number of solutions recorded
`valueSelection`	selects starting values of variables
`variableSelection`	selects which variables to start with
`assignOption`	selects which variables to assign values

reqT Entities can have a Constraints attribute containing a sequence of constraints.

```
var myReqs = Model(
  Feature("nice") has Spec("this is a nice feature"),
  Feature("cool") has Spec("this is a cool feature"),
  Stakeholder("Anna") has Constraints(
    (Feature("nice")!Prio) #< 10,
    (Feature("nice")!Prio) #>= 1,
    (Feature("cool")!Prio)::{2 to 7}
  ),
  Stakeholder("Martin") has Constraints(
    (Feature("nice")!Prio) #< 3,
    (Feature("nice")!Prio) #!= 1,
    (Feature("cool")!Prio)::{5 to 10}
  )
)
```

reqT Release Planning Input Data Model

```
val m = Model(
  Stakeholder("kalle") has (Prio(10), Submodel(
    Feature("F1") has Benefit(20),
    Feature("F2") has Benefit(20),
    Feature("F3") has Benefit(20)
  )),
  Stakeholder("stina") has (Prio(20), Submodel(
    Feature("F1") has Benefit(5),
    Feature("F2") has Benefit(15),
    Feature("F3") has Benefit(35)
  )),
  Resource("developer") has Submodel(
    Release("a") has Capacity(100),
    Release("b") has Capacity(100),
    Feature("F1") has Cost(10),
    Feature("F2") has Cost(70),
    Feature("F3") has Cost(20)
  ),
  Resource("tester") has Submodel(
    Release("a") has Capacity(100),
    Release("b") has Capacity(100),
    Feature("F1") has Cost(40),
    Feature("F2") has Cost(10),
    Feature("F3") has Cost(50)
  ),
  Release("a") has Order(1),
  Release("b") has Order(2)
)
```

reqT Release Planning: Vectors of Input Entities to prepare imposed constraints

```
val features = (m.flatten / Feature).sourceVector
val releases = (m / Release).sourceVector
val resources = (m / Resource).sourceVector
val stakeholders = (m / Stakeholder).sourceVector

val constraints = ???    // to be defined
val utility =  ???       // to be defined

val (m2, r) = Model().impose(constraints).solve(Maximize(utility))
```

reqT Release Planning: Assign values from Model

The XeqC constraint can be constructed by the #== infix operator on Var. Example of how to make a sequence of constraints that grounds integer variables to release planning input data from a model:

```
def assignValuesFromModel(m: Model) = Constraints(
  forAll(stakeholders) { s => Var(s!Prio) #== m(s!Prio) }  ++
  forAll(releases) { r => Var(r!Order) #== m(r!Order) } ++
  forAll(stakeholders, features) {
    (s,f) => Var(s!f!Benefit) #== m(s!f!Benefit) } ++
  forAll(resources, features) {
    (res, f) => Var(res!f!Cost) #== m(res!f!Cost) } ++
  forAll(resources, releases) {
    (res, r) => Var(res!r!Capacity) #== m(res!r!Capacity) }
)
```

reqT Release Planning Constraints 1(9)

All Features shall have an Order integer attribute to model that it can be allocated to some Release (corresponding to the Order attribute of that Release).

```
forAll(features) { f => (f!Order)::{1 to releases.size} }
```

reqT Release Planning Constraints 2(9)

For all stakeholders s and all features f:
Var(benefit(s,f)) is the benefit of the feature according to that stakeholder multiplied with the priority of the stakeholder.

```
forAll(stakeholders, features) { (s, f) =>
  (s!f!Benefit) * (s!Prio) #== Var(s"benefit($s,$f)")
}
```

reqT Release Planning Constraints 3(9)

For all features f:
Var(benefit(f)) is equal to the sum of all stakeholders' benefits of that f

```
forAll(features) { f =>
  sumForAll(stakeholders)( s => Var(s"benefit($s,$f)")) #==
    Var(s"benefit($f)")
}
```

reqT Release Planning Constraints 4(9)

for all releases r, for all features f:
if f is allocated to r **then** $\textit{benefit}(r, f)) = \textit{benefit}(f)$
else $\textit{benefit}(r, f)) = 0$

```
forAll(releases, features) { (r, f) =>
  IfThenElse(Var(f!Order) #== (r!Order),
    Var(s"benefit($r,$f)") #== Var(s"benefit($f)"),
    Var(s"benefit($r,$f)") #== 0)
}
```

reqT Release Planning Constraints 5(9)

For all releases r:
totBenefit(f) is the sum of all features' benefits of that r

```
forAll(releases) { r =>
  sumForAll(features)(f => Var(s"benefit($r,$f)")) #==
    Var(s"totBenefit($r)")
}
```

reqT Release Planning Constraints 6(9)

For all releases rel, for all features f, for all resources res:
If f is allocated to rel then `cost(rel, f, res)` is the cost of that feature needed by that resource, else it is zero.

```
forAll(releases,features, resources) { (rel, f, res) =>
  IfThenElse((f!Order) #== (rel!Order),
    Var(s"cost($rel,$f,$res)") #== (res!f!Cost),
    Var(s"cost($rel,$f,$res)") #== 0)
}
```

reqT Release Planning Constraints 7(9)

For all resources res, for all releases rel, for all features f:
totCost(rel, res) is the sum over all features of cost(rel, f, res)

```
forAll(resources, releases) { (res, rel) =>
  sumForAll(features)(f => Var(s"cost($rel,$f,$res)")) #==
    Var(s"totCost($rel,$res)")
}
```

reqT Release Planning Constraints 8(9)

For all resources res, for all releases rel:
totCost(res, rel) must be lest than or equal to the available capacity

```
forAll(resources,releases) { (res, rel) =>
  Var(s"totCost($rel,$res)") #<= (res!rel!Capacity)
}
```

reqT Release Planning Constraints 9(9)

For all releases, for all resources:
the total cost of release is the sum of the totalCost of all resources for that release

```
forAll(releases) { rel =>
  sumForAll(resources)(res => Var(s"totCost($rel,$res)")) #==
    Var(s"totCost($rel)")
}
```

reqT Release Planning: All 9 Constraints

```
val releasePlanningConstraints =  Constraints(
  forAll(features) { f => (f!Order)::{1 to releases.size} } ++
  forAll(stakeholders, features) { (s, f) =>
    (s!f!Benefit) * (s!Prio) #== Var(s"benefit($s,$f)") } ++
  forAll(features) { f =>
    sumForAll(stakeholders)(s => Var(s"benefit($s,$f)")) #== Var(s"benefit($f)") } ++
  forAll(releases, features) { (r, f) =>
    IfThenElse((f!Order) #== (r!Order),
      Var(s"benefit($r,$f)") #== Var(s"benefit($f)"),
      Var(s"benefit($r,$f)") #== 0) } ++
  forAll(releases) { r =>
    sumForAll(features)(f => Var(s"benefit($r,$f)")) #== Var(s"totBenefit($r)") } ++
  forAll(releases,features, resources) { (rel, f, res) =>
    IfThenElse((f!Order) #== (rel!Order),
      Var(s"cost($rel,$f,$res)") #== (res!f!Cost),
      Var(s"cost($rel,$f,$res)") #== 0) } ++
  forAll(resources, releases) { (res, rel) =>
    sumForAll(features)(f => Var(s"cost($rel,$f,$res)")) #== Var(s"totCost($rel,$res)") } ++
  forAll(resources,releases) { (res, rel) =>
    Var(s"totCost($rel,$res)") #<= (res!rel!Capacity) } ++
  forAll(releases) { rel =>
    sumForAll(resources)(res => Var(s"totCost($rel,$res)")) #== Var(s"totCost($rel)") }
)
```

reqT Release Planning Optimization

```
val constraints =
  assignValuesFromModel(m) ++
  releasePlanningConstraints
val utility = Var("totBenefit(Release(a))")
val (m2, r) =
  Model().impose(constraints).solve(Maximize(utility))
```

```
reqT> val allocationModel = m2 / Feature
allocationModel: reqt.Model =
Model(
  Feature("F3") has Order(1),
  Feature("F1") has Order(2),
  Feature("F2") has Order(2)
)

reqT> val cost = r.lastSolution(Var("totCost(Release(a))"))
cost: Int = 70
```

reqT Release Planning:
Adding coupling and precedence constraints

Coupling: Two features must be in the same release:

```
(Feature("F1")!Order) #== (Feature("F2")!Order)
```

Precedence:
One features must be implemented before another feature:

```
(Feature("F2")!Order) #< (Feature("F3")!Order)
```

```
Model(
  Stakeholder("SiREN geek") requires Quality("feedback")
)
```

Conclusions and Discussion

Some results so far:

- Graph-oriented DSL for requirements embedded in Scala in place
- Entities, attributes and relations implemented for some (the most?) essential RE concepts
- Web document generation
- Export/import to spread sheet programs
- Integration with testing
- Integration with constraint solving for prioritization and release planning

Outlook on future work:

- Combining support for the QUPER model for QR with CSP
- Utilize research results in Natural Language Processing for RE
- Interface to visual graph generators
- Documentation and teaching material
- Product Line Engineering concepts and constraints of PLE feature models
- More complete implementation of JaCoP 4.0 constraints
- GUI support (MSc Thesis: Oskar Präntare & Joel Johansson)
- Investigation of how to utilize cutting-edge constraint solving technology:
  - Soft constraints
  - Stochastic constraints

Early adopters, contributers, independent assessments and feedback are sincerely welcome!

http://reqT.org